Tackling Gootkit's Traps

f5.com/labs/articles/threat-intelligence/tackling-gootkit-s-traps

byline

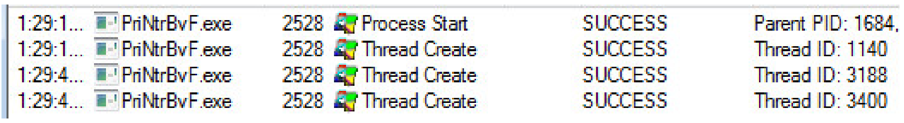
July 11, 2018

Gootkit is an advanced banking trojan first discovered in mid-2014. Known for using various techniques to evade detection, the malware also has its own unique twists: it’s partially written in JavaScript and it incorporates the node.js runtime environment.

Gootkit employs several checks in order to identify a virtual environment and halt its propagation sequence once that happens. In this post, we’ll demonstrate a way to tackle these anti-research methods in order to have the sample executing in a virtual environment.

First Glance

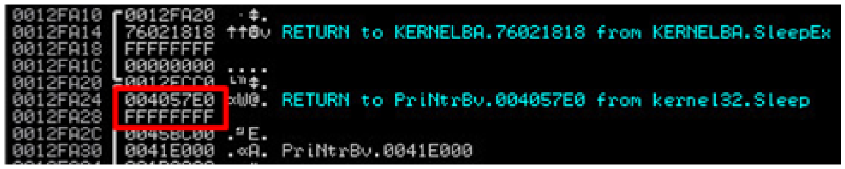
Running the sample in a virtual machine results in the creation of several threads, but no modifications to the system are made. Without interruptions, Gootkit’s process would continue to run forever.



```
1:29:1...  PriNtrBvF.exe  2528  Process Start   SUCCESS  Parent PID: 1684,
1:29:1...  PriNtrBvF.exe  2528  Thread Create   SUCCESS  Thread ID: 1140
1:29:4...  PriNtrBvF.exe  2528  Thread Create   SUCCESS  Thread ID: 3188
1:29:4...  PriNtrBvF.exe  2528  Thread Create   SUCCESS  Thread ID: 3400
```

Figure 1: procmon view of the sample

Running the sample in a debugger allows us to interrupt the process and inspect its execution flow. Whenever the debugger is paused, Gootkit’s process is sleeping.



```
0012FA10  0012FA20
0012FA14  76021818  RETURN to KERNELBA.76021818 from KERNELBA.SleepEx
0012FA18  FFFFFFFF
0012FA1C  00000000
0012FA20  0012FCC0
0012FA24  004057E0  RETURN to PriNtrBv.004057E0 from kernel32.Sleep
0012FA28  FFFFFFFF
0012FA2C  0045BC00
0012FA30  0041E000  PriNtrBv.0041E000
```

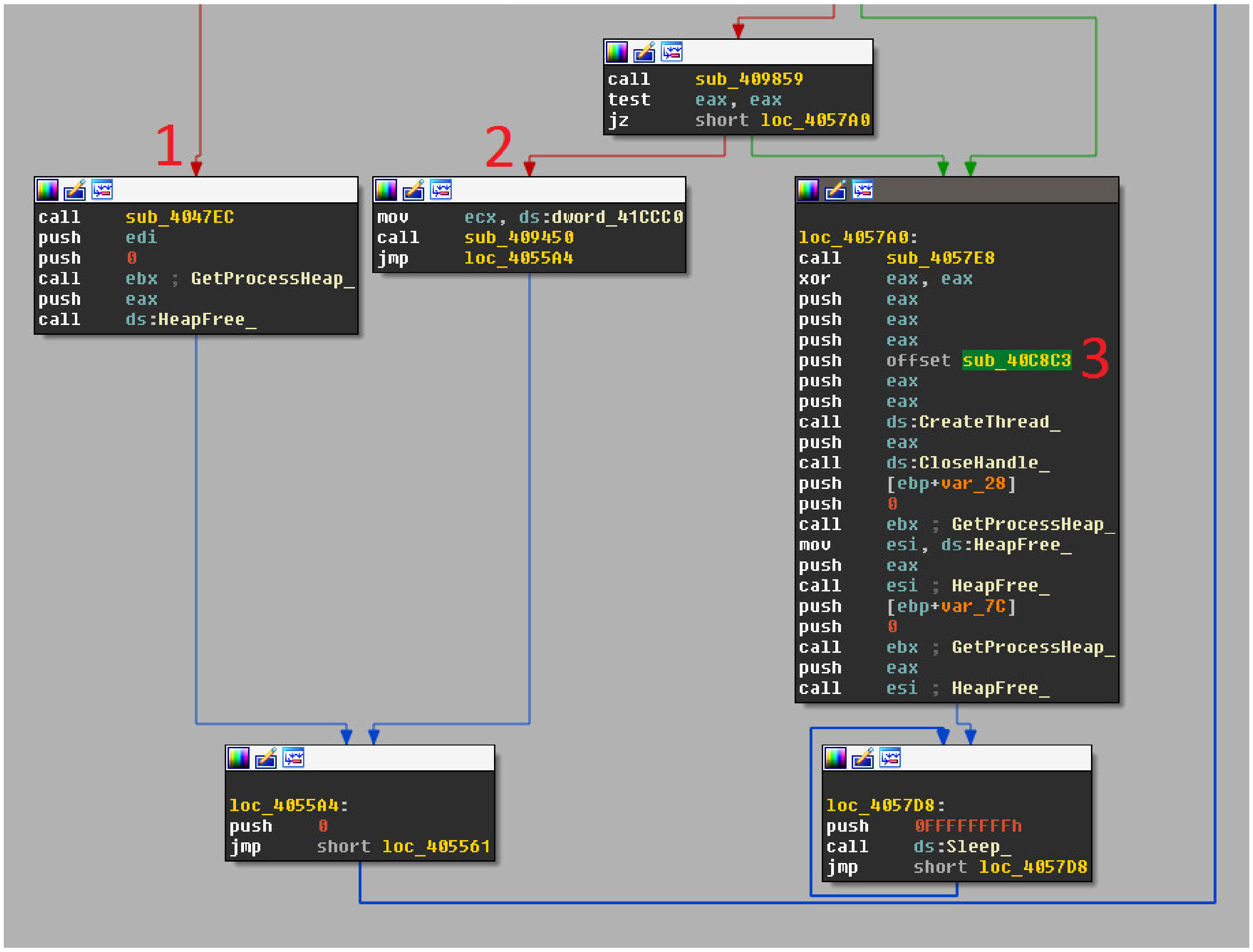
Figure 2: View of the stack of the sleeping thread

According to the return address, the call comes from the malware (not from a system library) and the parameter is -1, meaning: sleep forever.

How can we make the malware go to a more interesting execution flow in the code?

Diving Deeper into the Assembly Code

Looking at the assembly of the function that calls the sleep API, there are three possible execution paths: #1 and #2 are paths that we could force the malware to take by patching the conditional jumps that lead to them; #3 is the path it takes before reaching the sleep API.



```
call    sub_409859
test    eax, eax
jz      short loc_4057A0
```

```
call    sub_4047EC
push    edi
push    0
call    ebx ; GetProcessHeap_
push    eax
call    ds:HeapFree_
```

```
mov     ecx, ds:dword_41CCC0
call    sub_409450
jmp     loc_4055A4
```

```
loc_4057A0:
call    sub_4057E8
xor     eax, eax
push    eax
push    eax
push    eax
push    offset sub_40C8C3
push    eax
push    eax
call    ds:CreateThread_
push    eax
call    ds:CloseHandle_
push    [ebp+var_28]
push    0
call    ebx ; GetProcessHeap_
mov     esi, ds:HeapFree_
push    eax
call    esi ; HeapFree_
push    [ebp+var_7C]
push    0
call    ebx ; GetProcessHeap_
push    eax
call    esi ; HeapFree_
```

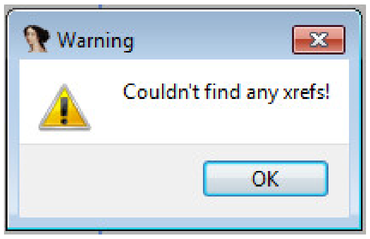
```
loc_4055A4:
push    0
jmp     short loc_405561
```

```
loc_4057D8:
push    0FFFFFFFFh
call    ds:Sleep_
jmp     short loc_4057D8
```

Figure 3: The calling function of the sleep API, thread 1

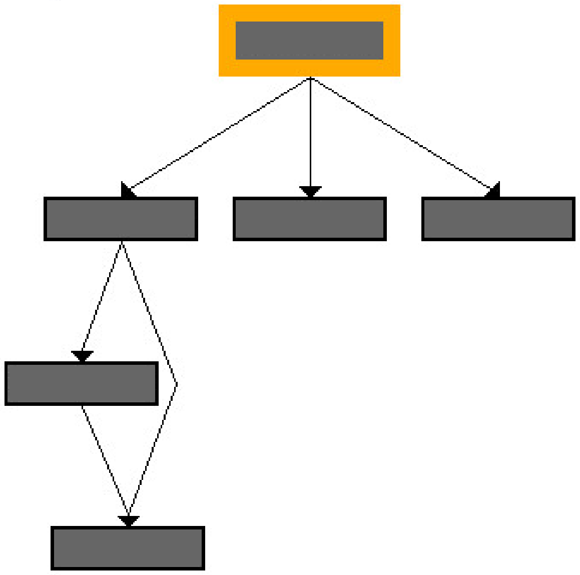
There are 3 possible paths of execution. We'll use a shortcut in order to decide quickly which one is the most interesting one to follow. That shortcut involves checking cross-references(xrefs) from each function, which will indicate where most of the activity would take place.

Execution path #1, contains a call to sub_4047EC, which had no xrefs.

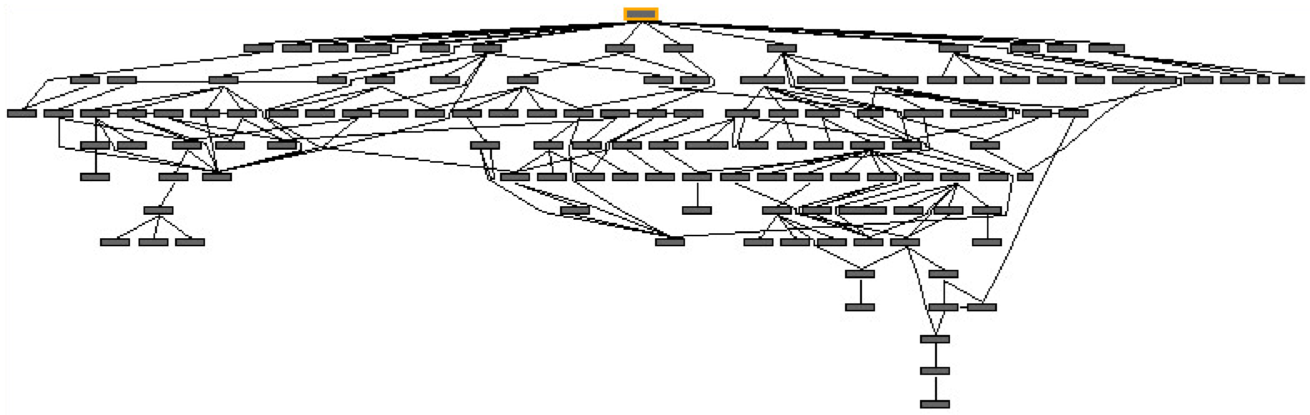

Warning

Couldn't find any xrefs!

Execution path #2, contains a call to sub_409450, which had just a few xrefs.



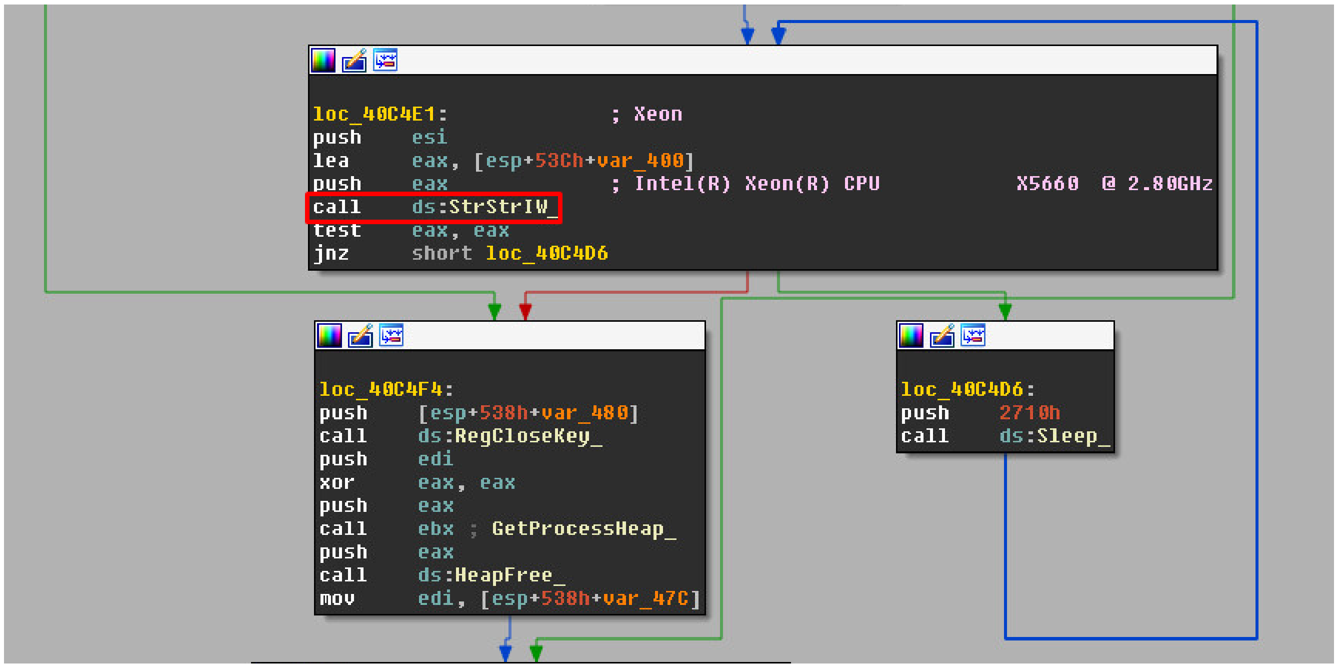
Execution path #3 creates a thread that will execute sub_40C8C3, which had many xrefs.



So, execution path #3, the newly created thread, seems like the one to take because it has a massive function graph, indicating that all the activity will most likely happen there.

After placing a breakpoint at the beginning of the thread function(0x40C8C3), we start stepping through. One of the subroutines gets stuck in an endless loop, which means Gootkit is sleeping again!

Recap #1: There are two threads, one of which is sleeping forever, but it is not the interesting one. The second one is in a sleep loop, and our goal is to force it to take a different branch in order to execute the interesting functionality.



```
loc_40C4E1:                 ; Xeon
push    esi
lea     eax, [esp+53Ch+var_400]
push    eax                 ; Intel(R) Xeon(R) CPU           X5660  @ 2.80GHz
call    ds:StrStrIW_
test    eax, eax
jnz     short loc_40C4D6
```

```
loc_40C4F4:
push    [esp+538h+var_480]
call    ds:RegCloseKey_
push    edi
xor     eax, eax
push    eax
call    ebx ; GetProcessHeap_
push    eax
call    ds:HeapFree_
mov     edi, [esp+538h+var_47C]
```

```
loc_40C4D6:
push    2710h
call    ds:Sleep_
```

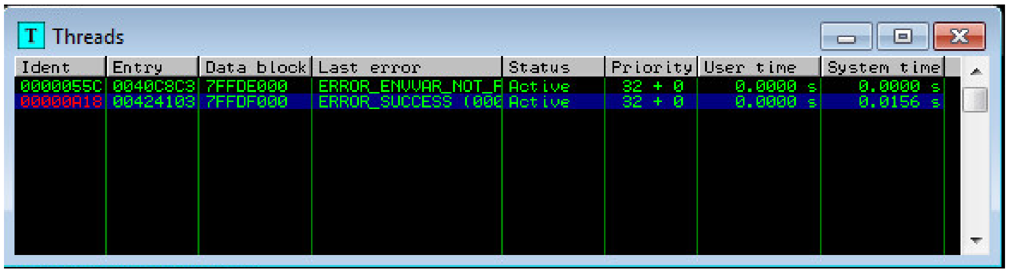
Figure 4: The calling function of the sleep API, thread #2

The only way to break the loop is to make the string comparison(StrStrIW) fail. The comparison parameters:

- Hard-coded string “Xeon”
- Data from *HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0\ProcessorNameString*

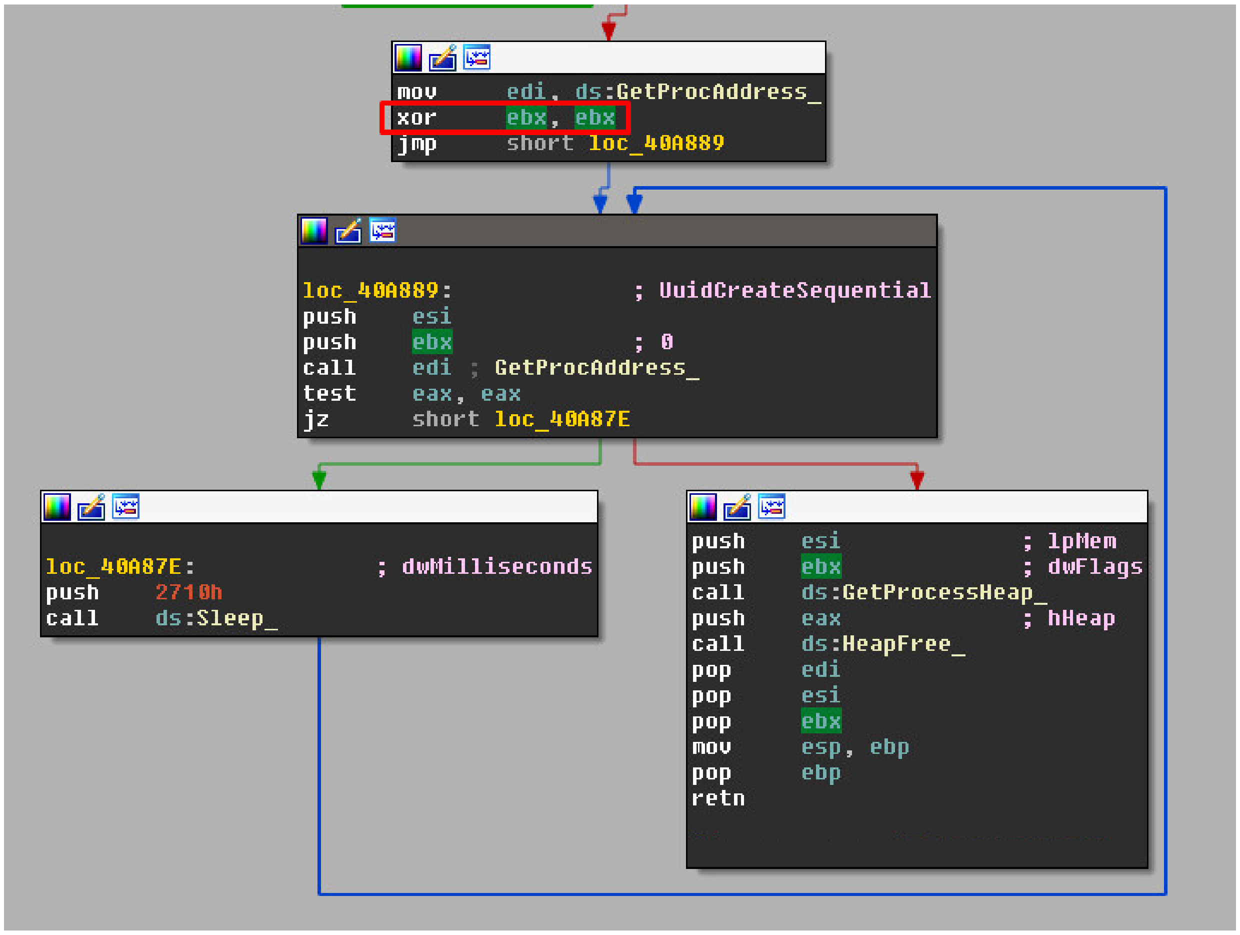
If we changed this value in the registry, we could successfully overcome this hurdle. But, would that be the end? Would it run? Sadly, after making this change, Gootkit goes back to sleep in another endless loop, this time in a different subroutine.

Recap #2: Again, there are two threads, one that is sleeping forever (but not interesting); the other is running in a sleep loop.



Ident	Entry	Data block	Last error	Status	Priority	User time	System time
0000055C	0040C8C3	7FFDE000	ERROR_ENVVAR_NOT_F	Active	32 + 0	0.0000 s	0.0000 s
00000A18	00424103	7FFDF000	ERROR_SUCCESS (000	Active	32 + 0	0.0000 s	0.0156 s

Figure 5: Gootkit’s two threads in Immunity Debugger’s thread view



```
mov     edi, ds:GetProcAddress_
xor     ebx, ebx
jmp     short loc_40A889
```

```
loc_40A889:                 ; UuidCreateSequential
push    esi
push    ebx                 ; 0
call    edi ; GetProcAddress_
test    eax, eax
jz      short loc_40A87E
```

```
loc_40A87E:                 ; dwMilliseconds
push    2710h
call    ds:Sleep_
```

```
push    esi                 ; lpMem
push    ebx                 ; dwFlags
call    ds:GetProcessHeap_
push    eax                 ; hHeap
call    ds:HeapFree_
pop     edi
pop     esi
pop     ebx
mov     esp, ebp
pop     ebp
retn
```

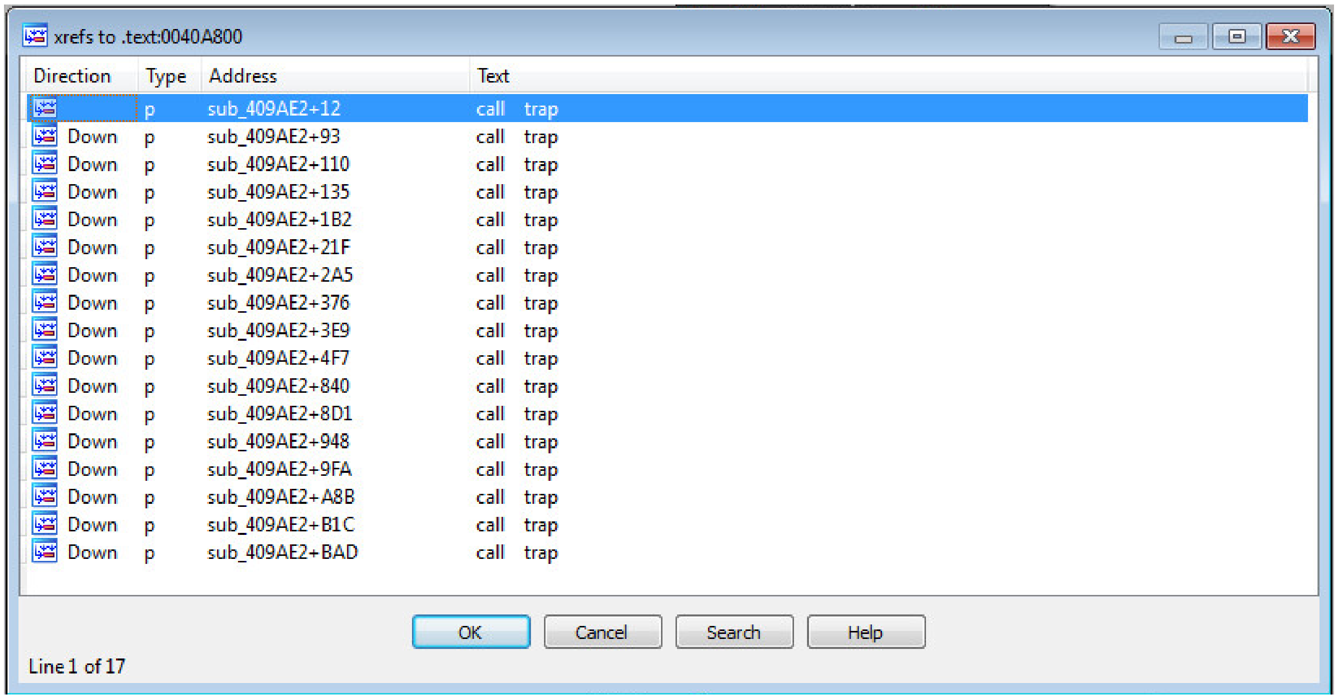
Figure 6: Another calling function of the sleep API, thread #2

After inspecting the condition that leads us to the sleep API call, it becomes clear that this branch is taken when the “GetProcAddress” fails. But, why would this simple API fail?

At second glance, it is evident that the first parameter of the “GetProcAddress” API, the HModule, is always equal to null because of the XOR instruction highlighted in red. The conclusion is that this API is meant to fail! This whole function is a trap, and our goal is to avoid entering it altogether.

Following the Traps

We’ll rename the function to “trap” and then check all the function references to see how many traps are there.


xrefs to .text:0040A800

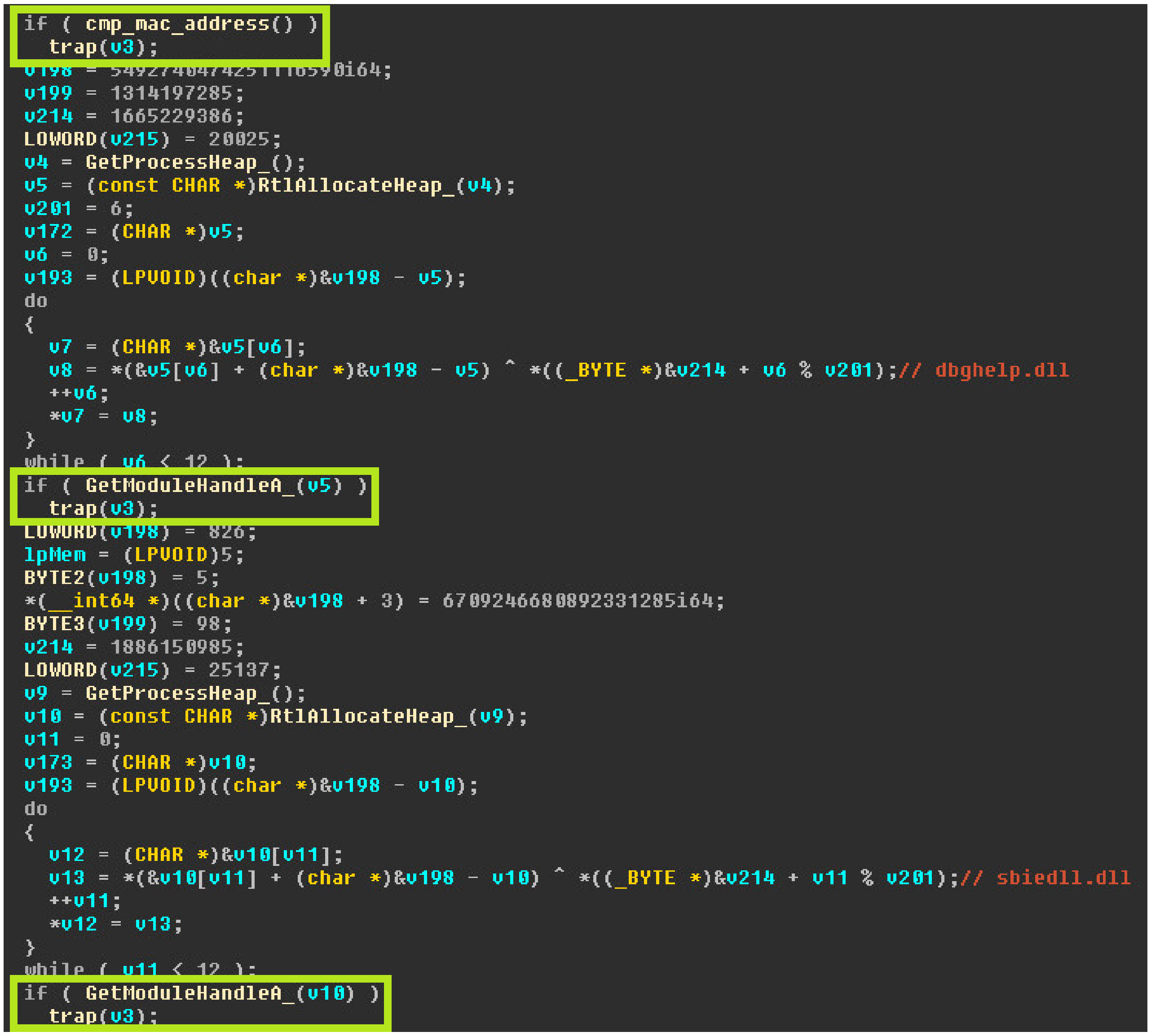
Direction	Type	Address	Text
	p	sub_409AE2+12	call trap
Down	p	sub_409AE2+93	call trap
Down	p	sub_409AE2+110	call trap
Down	p	sub_409AE2+135	call trap
Down	p	sub_409AE2+1B2	call trap
Down	p	sub_409AE2+21F	call trap
Down	p	sub_409AE2+2A5	call trap
Down	p	sub_409AE2+376	call trap
Down	p	sub_409AE2+3E9	call trap
Down	p	sub_409AE2+4F7	call trap
Down	p	sub_409AE2+840	call trap
Down	p	sub_409AE2+8D1	call trap
Down	p	sub_409AE2+948	call trap
Down	p	sub_409AE2+9FA	call trap
Down	p	sub_409AE2+A8B	call trap
Down	p	sub_409AE2+B1C	call trap
Down	p	sub_409AE2+BAD	call trap

Line 1 of 17

Figure 7: All function references to the trap function

There are 17 traps to overcome and they are all in the same function, so we assume this is the function making all the environment checks. We'll have to check each condition that leads to a trap and make sure to change the environment (or patch the binary) in a way that would bypass the trap.

Figure 8 shows a code snippet from the sub_409AE2 function.



```
if ( cmp_mac_address() )
  trap(v3);
v198 = 5492740474251116590i64;
v199 = 1314197285;
v214 = 1665229386;
LOWORD(v215) = 20025;
v4 = GetProcessHeap_();
v5 = (const CHAR *)RtlAllocateHeap_(v4);
v201 = 6;
v172 = (CHAR *)v5;
v6 = 0;
v193 = (LPVOID)((char *)&v198 - v5);
do
{
  v7 = (CHAR *)&v5[v6];
  v8 = *(&v5[v6] + (char *)&v198 - v5) ^ *((_BYTE *)&v214 + v6 % v201);// dbghelp.dll
  ++v6;
  *v7 = v8;
}
while ( v6 < 12 );
if ( GetModuleHandleA_(v5) )
  trap(v3);
LOWORD(v198) = 826;
lpMem = (LPVOID)5;
BYTE2(v198) = 5;
*(__int64 *)((char *)&v198 + 3) = 6709246680892331285i64;
BYTE3(v199) = 98;
v214 = 1886150985;
LOWORD(v215) = 25137;
v9 = GetProcessHeap_();
v10 = (const CHAR *)RtlAllocateHeap_(v9);
v11 = 0;
v173 = (CHAR *)v10;
v193 = (LPVOID)((char *)&v198 - v10);
do
{
  v12 = (CHAR *)&v10[v11];
  v13 = *(&v10[v11] + (char *)&v198 - v10) ^ *((_BYTE *)&v214 + v11 % v201);// sbiedll.dll
  ++v11;
  *v12 = v13;
}
while ( v11 < 12 );
if ( GetModuleHandleA_(v10) )
  trap(v3);
```

Figure 8: The calling function for all the traps in pseudo C view

Note that each “while” loop is performing string decryption on the sequences of bytes shown in the variables above the loop. When following the execution in a debugger, the strings are decrypted, and some meaningful indicators of VM checks are visible. (See appendix for decryption function details.)

In this code snippet, three checks are evident:

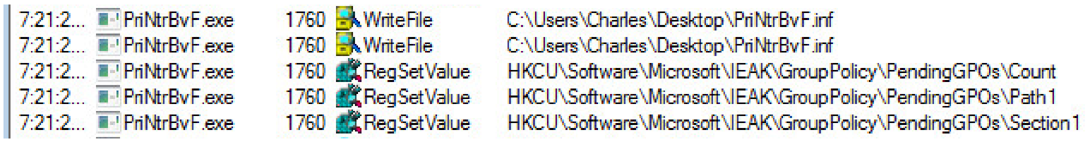
- MAC address check
- Checking the presence of “dbghelp.dll” — debugger indicator
- Checking the presence of “sbiedll.dll” — sandboxie indicator

By following the traps and patching the system accordingly, the environment is prepared for Gootkit to run in.

The rest of the checks include:

- Compare user name to "CurrentUser"/"Sandbox"
- Compare computer name to "SANDBOX"/"7SILVIA"
- *HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\SystemBiosVersion"* compare with AMI, VirtualBox, BOCHS, INTEL 640000, 55274-640-2673064-23950, and other serials

After patching a virtual machine and running the sample, it's clear that it is no longer stuck in an endless loop and that the sample continues its propagation in the system.



```
7:21:2... PriNtrBvF.exe  1760 WriteFile    C:\Users\Charles\Desktop\PriNtrBvF.inf
7:21:2... PriNtrBvF.exe  1760 WriteFile    C:\Users\Charles\Desktop\PriNtrBvF.inf
7:21:2... PriNtrBvF.exe  1760 RegSetValue  HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs\Count
7:21:2... PriNtrBvF.exe  1760 RegSetValue  HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs\Path1
7:21:2... PriNtrBvF.exe  1760 RegSetValue  HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs\Section1
```

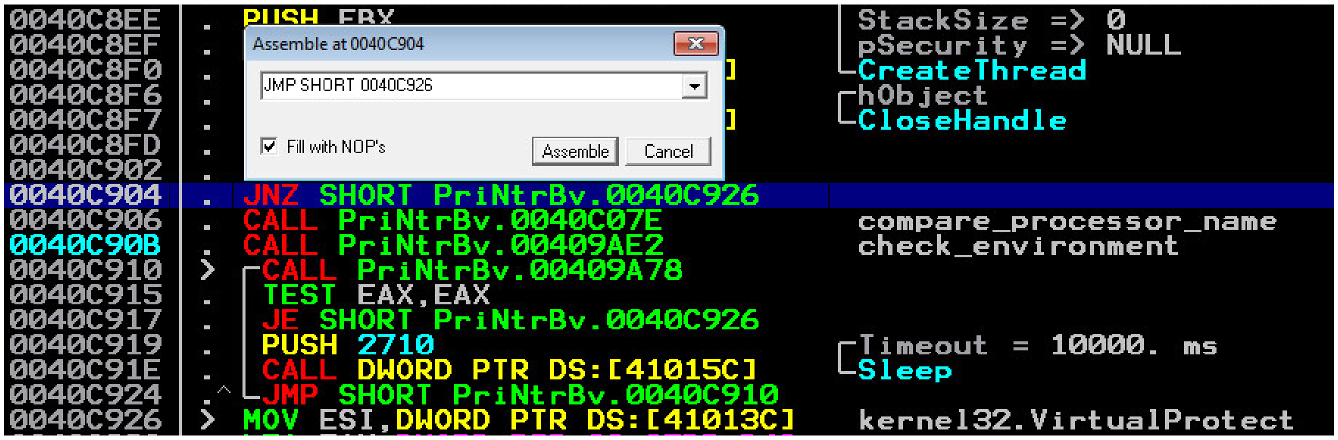
Figure 9: Gootkit installation process

The Road Not Taken

Patching the virtual environment by changing the MAC address and several registry values is one way to get the sample to run. It is the recommended way to go when running Gootkit samples in an automated analysis system. However, this post would not be complete without presenting the other solution: patching the binary itself.

Every time there's a conditional jump to the trap function, one could change it to a simple jump opcode, thus skipping the trap function. This would direct the flow from the jump to the next code block while skipping the trap function.

However, this patch must be applied 17 times, and that's a tedious task. Instead, since all the calls to the trap function come from one function, its preferable to patch and bypass the call to that function.



```
0040C8EE  .  PUSH EBX                                  StackSize => 0
0040C8EF  .                                            pSecurity => NULL
0040C8F0  .                                      ]     CreateThread
0040C8F6  .                                            hObject
0040C8F7  .                                      ]     CloseHandle
0040C8FD  .
0040C902  .
0040C904  .  JNZ SHORT PriNtrBv.0040C926
0040C906  .  CALL PriNtrBv.0040C07E                    compare_processor_name
0040C90B  .  CALL PriNtrBv.00409AE2                    check_environment
0040C910  >  CALL PriNtrBv.00409A78
0040C915  .  TEST EAX,EAX
0040C917  .  JE SHORT PriNtrBv.0040C926
0040C919  .  PUSH 2710                                 Timeout = 10000. ms
0040C91E  .  CALL DWORD PTR DS:[41015C]                Sleep
0040C924  .^ JMP SHORT PriNtrBv.0040C910
0040C926  >  MOV ESI,DWORD PTR DS:[41013C]             kernel32.VirtualProtect
```

Figure 10: The "JNZ" instruction opcode(0x75) is changed to the "JMP" instruction opcode(0xEB) in Immunity Debugger

Conclusion

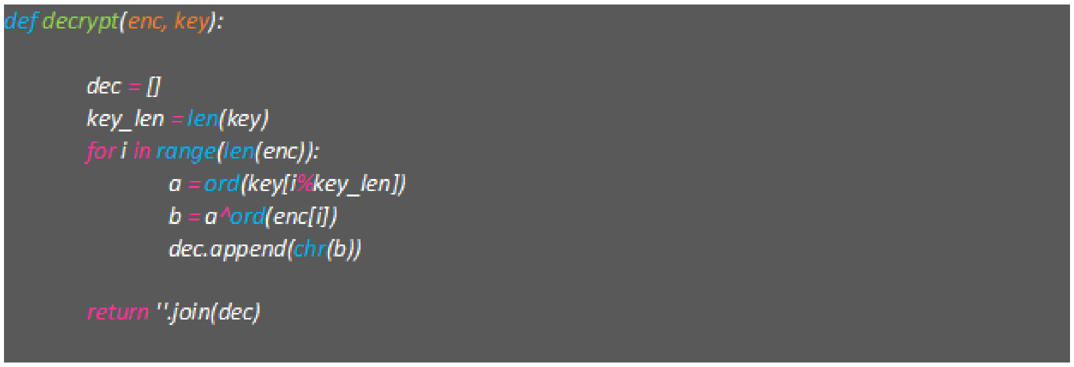
When analyzing malware, especially intricate, multi-threaded ones like Gootkit, it is important to choose the interesting thread to follow and to keep in mind that even functions that look like they contain seemingly "ordinary" code may be used as a deception.

MD5: 47f8d6d3e4410218c4c64701cb7d7d3d

Appendix

The decryption function is an XOR loop with a hard-coded key. The encrypted string and the key are hard-coded sequences of bytes.

In Python:

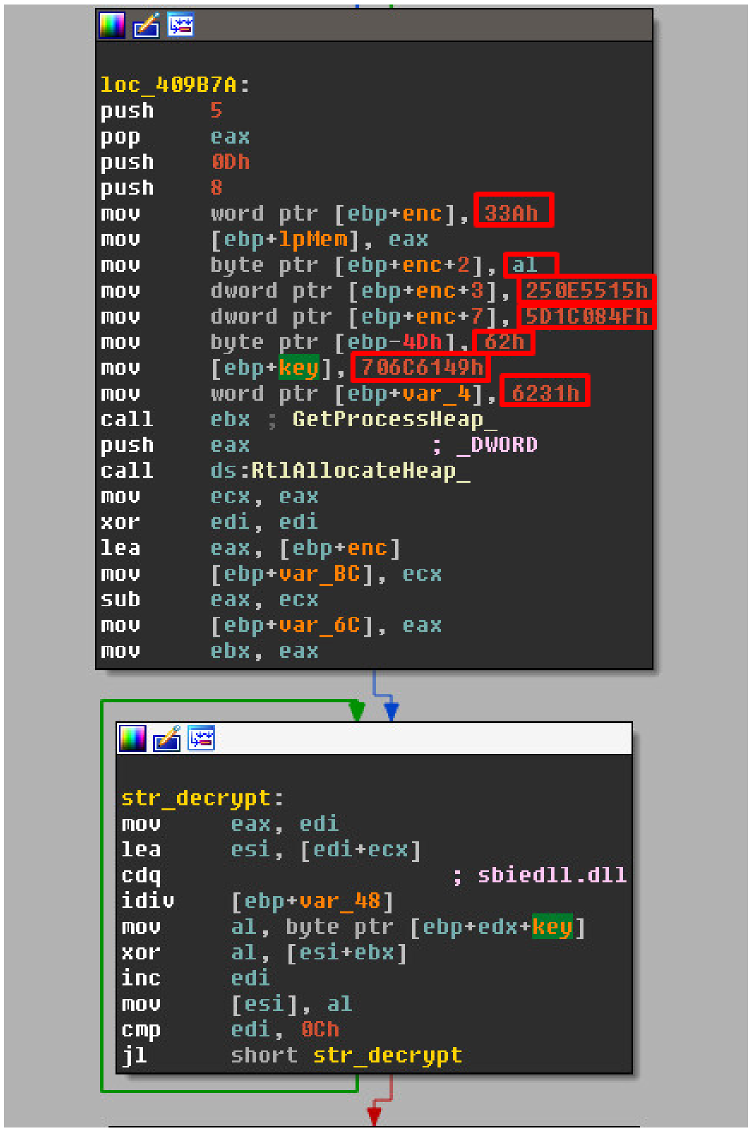


```python
def decrypt(enc, key):

    dec = []
    key_len = len(key)
    for i in range(len(enc)):
        a = ord(key[i%key_len])
        b = a^ord(enc[i])
        dec.append(chr(b))

    return ''.join(dec)
```

An example of the string decryption loop and its parameters:



```
loc_409B7A:
push    5
pop     eax
push    0Dh
push    8
mov     word ptr [ebp+enc], 33Ah
mov     [ebp+lpMem], eax
mov     byte ptr [ebp+enc+2], al
mov     dword ptr [ebp+enc+3], 250E5515h
mov     dword ptr [ebp+enc+7], 5D1C084Fh
mov     byte ptr [ebp-4Dh], 62h
mov     [ebp+key], 706C6149h
mov     word ptr [ebp+var_4], 6231h
call    ebx ; GetProcessHeap_
push    eax             ; _DWORD
call    ds:RtlAllocateHeap_
mov     ecx, eax
xor     edi, edi
lea     eax, [ebp+enc]
mov     [ebp+var_BC], ecx
sub     eax, ecx
mov     [ebp+var_6C], eax
mov     ebx, eax

str_decrypt:
mov     eax, edi
lea     esi, [edi+ecx]
cdq                     ; sbiedll.dll
idiv    [ebp+var_48]
mov     al, byte ptr [ebp+edx+key]
xor     al, [esi+ebx]
inc     edi
mov     [esi], al
cmp     edi, 0Ch
jl      short str_decrypt
```

encrypted_string = "\x3A\x03\x05\x15\x55\x0E\x25\x4F\x08\x1C\x5D\x62"

key = "\x49\x61\x6C\x70\x31\x62"